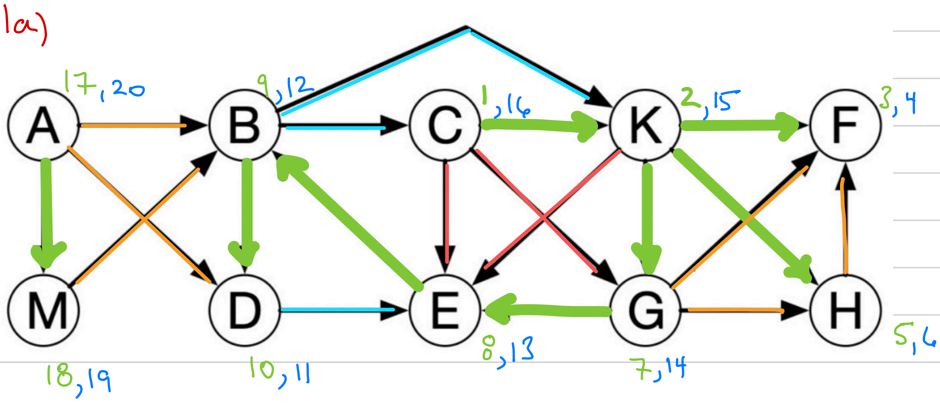
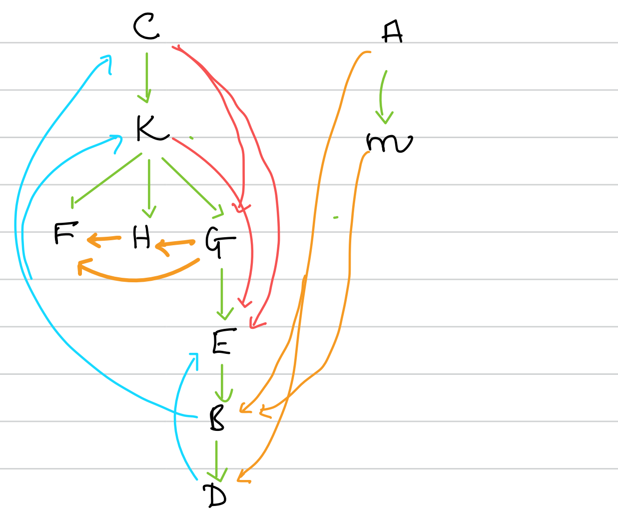
Assignment 5 solutions

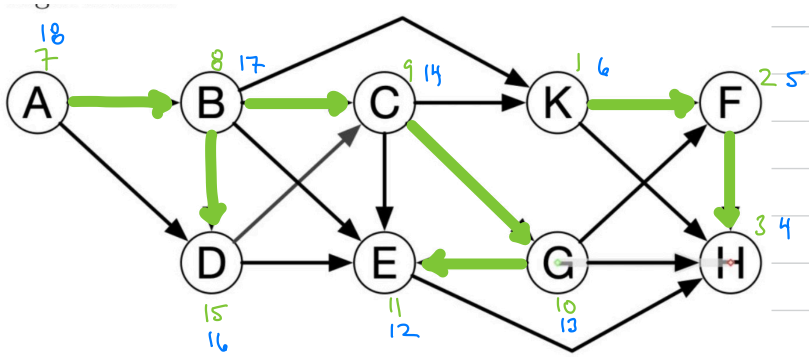
1a)

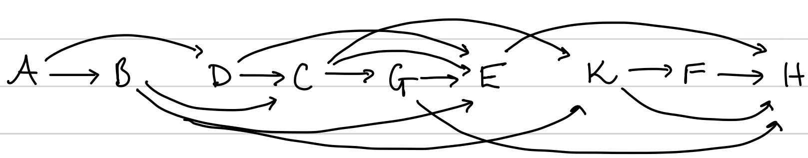




— Forward — Back . — Back

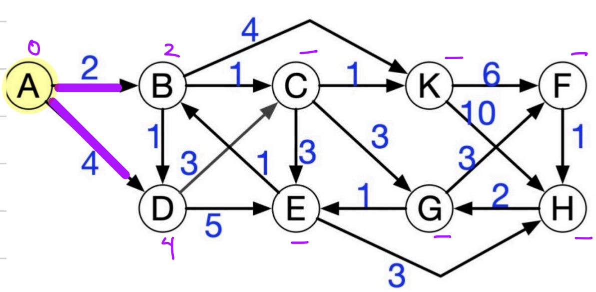
1b)



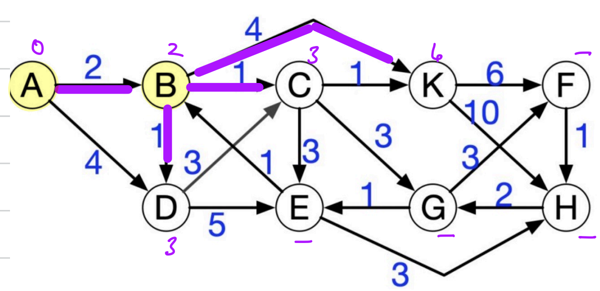


1c) yellow vertices have been removed from Q

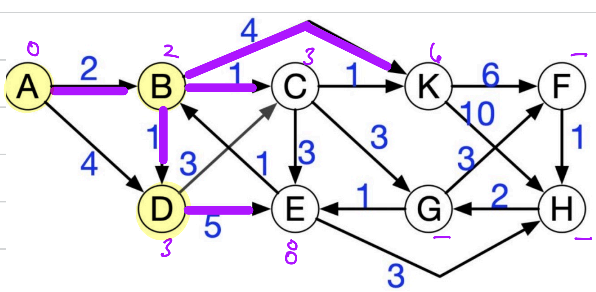
1)



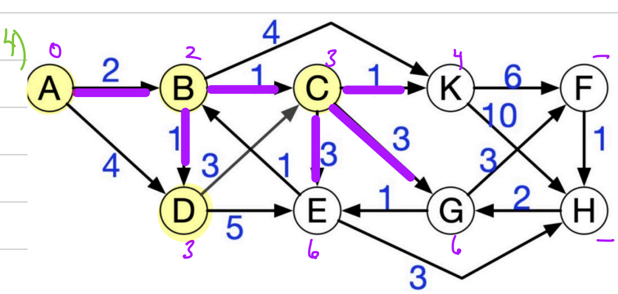
2)



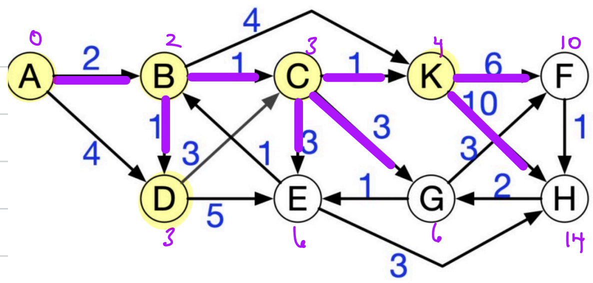
3)



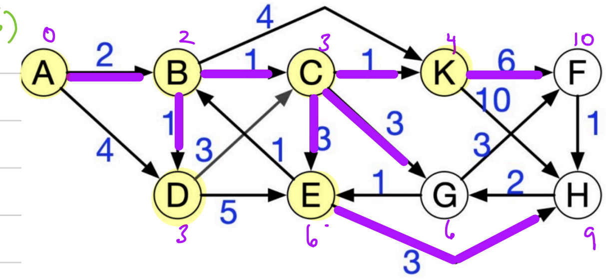
4)



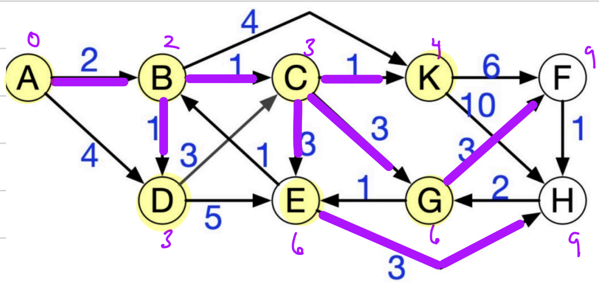
5)



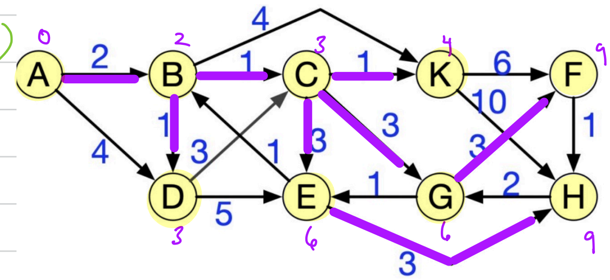
6)



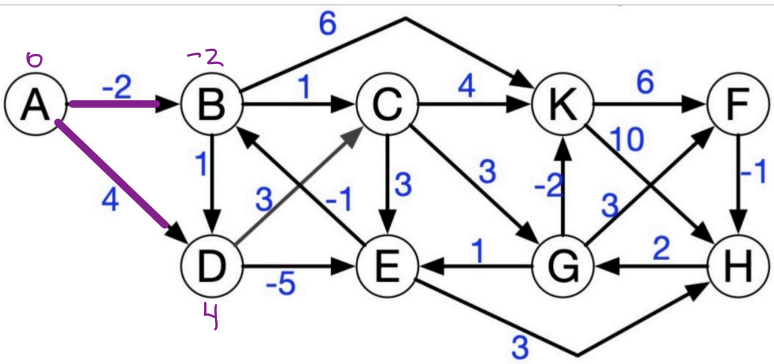
7)



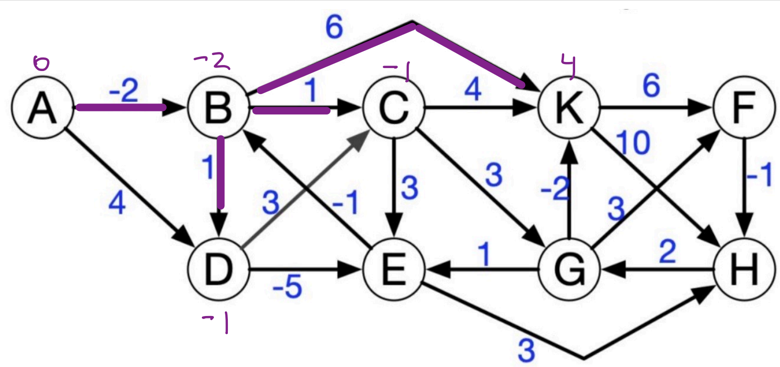
8)



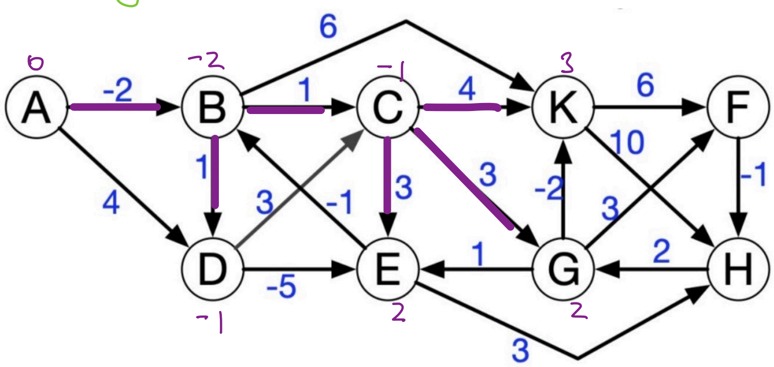
(e) 1) Edges out of A:



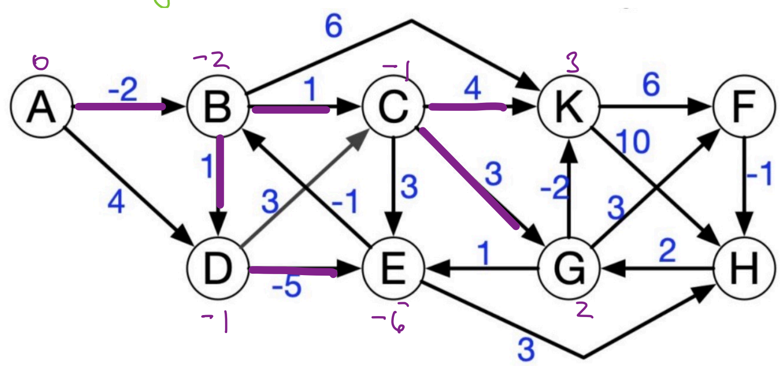
2) Edges out of B:



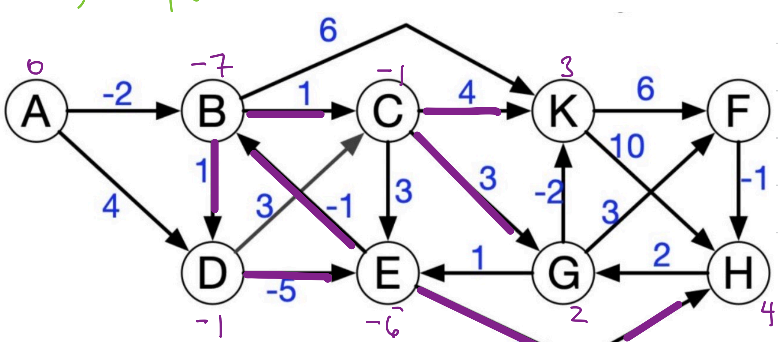
3) Edges out of C:



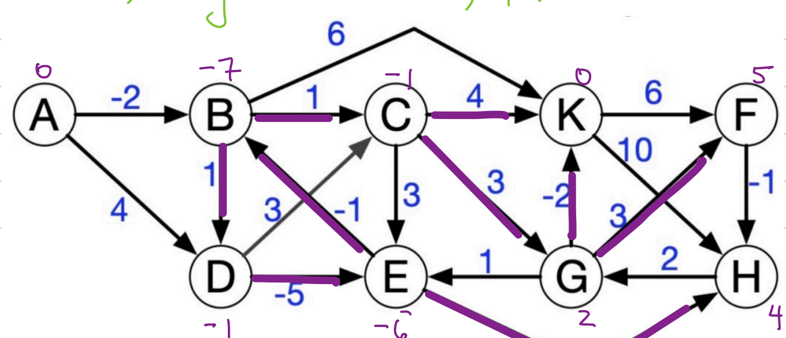
4) Edges out of D:



5) Edges out of E:



6) Edges out of F, G:



7) Edges out of H, K : no change // End of iteration 1.

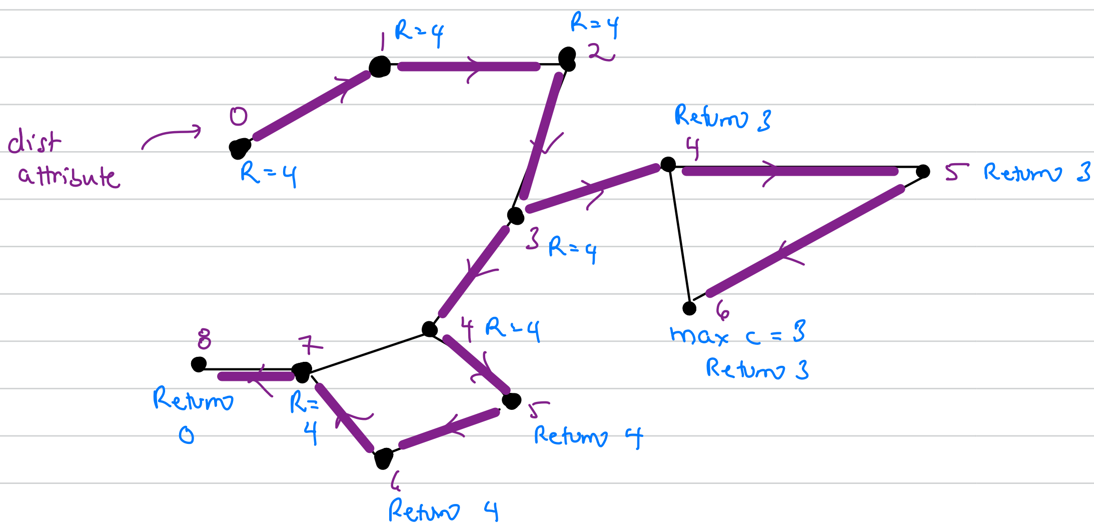
For each subsequent iteration, B.d will decrease by 5. Therefore at iteration 9 a neg weight cycle will be detected.

2a) Update DFS visit so that it returns the max. cycle length down the DFS tree from vertex u. Each time a NEW cycle is found, we compare its length to the current max.

Since this is a simple (constant-time) change to DFS, the runtime is $O(V+E)$.

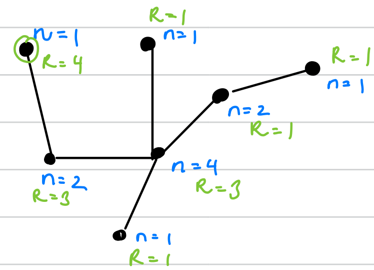
Initialise: Start at any vertex s, set s.dist = 0

```
MaxCycle(u)
    maxc = 0
    u.visited = TRUE
    for all v in Adj(u)
        if v.visited = False
            v.parent = u
            v.dist = u.dist + 1
            maxc = max(maxc, MaxCycle(v))   // update max
        else if v ≠ u.parent   // found backedge = cycle
            L = u.dist - v.dist + 1   // new cycle length
            maxc = max(L, maxc)
    Return maxc.
```



b) Start the procedure below using any vertex u from V. The constant-time change to DFS means that the runtime is O(V+E)

```
CountLeaves (u)
    u.visited = TRUE
    count = 0
    Neighbors = 0
    for all v in Adj(u)
        Neighbours ++
        if v.visited = False
            v.parent = u
            count += CountLeaves (v)
    if Neighbours = 1
        Return 1 + count
    else Return count.
```



c) Model the graph as a directed unweighted graph, where vertex = farm, and directed edge is one way road. Additionally, each vertex has attribute v.scc to identify which component its in.

Step 1) Run DFS(G) on original graph, where DFS-visit adds terminated vertices to the back of list L.

```
DFS-visit(u)
    u.visited = TRUE
    for all v in Adj(u)
        if v.visited = False
            DFS-visit(v)
    L.add(u)
```

Now L is a list of vertices in decreasing order of finish time.

Runtime: O(V+E)

Step 2) Create new graph G^T using edge set E^T which contains the original edges from E but in reverse order. O(E) for reversing edges.

Step 3) Use first vertex from L and run DFS-visit(u) with the following update:

```
DFS-visit(u)
    u.visited = TRUE
    u.scc = 1
    for all v in Adj(u)
        if v.visited = False
            DFS-visit(u)
```

O(V+E)

Now each vertex in SCC 1 is indicated by attribute u.scc = 1.

Step 4) Find another vertex that is part of the 2nd SCC by finding an unvisited vertex.

```
for all v in V
    if v.visited = False
        s = v
        break
```

O(V)

Step 5) Loop through the vertices of SCC1 and identify a vertex that can be connected to S. Note that since S in SCC2, we know SCC1 → SCC2, but not the reverse.

```
for all v in V
    if v.scc = 1
        if (v,s) ∉ E
            Print add edge (s,v) to G.
            break
```

$\left.\right\}$ O(V)

Runtime: $O(V+E) = O(n+n^2) = O(n^2)$ since |E| is

(d) Step 1) Run DFS from S, using only vertices for which $b(u,v) = false$.
Use visited attribute v.fromS

```
DFS-visit(u)
    u.fromS
    for all v in Adj(u)
        if b(u,v) = false   //not a bridge
            if v.fromS = false
                v.parent = u
                DFS-visit(v)
```

Step 2) Now all visited vertices are those we can reach from S without using a bridge. If T.fromS = T, return true.

Step 3) Run DFS from vertex T, again using only edges $b(u,v) = false$.
Use visited attribute v.fromT

Step 4) Find if there is 1 bridge connected each path:

```
for all e=(u,v) in E
    if b(u,v) = TRUE   //bridge
        if u.fromS = T and v.fromT = T
            return TRUE
Return False
```

Runtime: Each DFS runs in $O(V+E)$ and step 4 in time $O(E)$.
∴ total runtime $O(V+E)$.

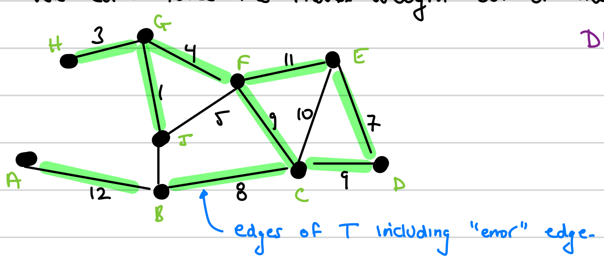
(e) Update Step 2 above: if T.fromS = T then call PrintBackwards(T)
else ... in Step 4:

```
for all e = (u,v) in E
    if b(u,v) = TRUE
        if u.fromS = T and v.fromT = T
            PrintBackwards(u)
            PrintForwards(v)
            Return TRUE
```

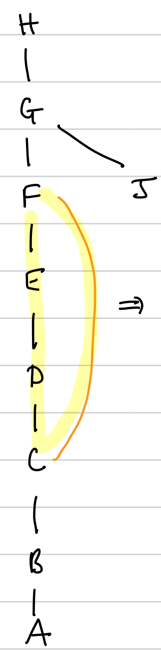
PrintBackwards(u)
    if u ≠ NIL
        PrintBackwards(u.parent)
        Print u

PrintForwards(u)
    if u ≠ NIL
        Print u
        PrintForwards(u.parent)

3a) Create a new graph G' with vertex set V and edges the set of edges in T (including the error). $|T| = O(V)$.
Note that the "error" edge must have created a cycle. We can run Find Cycle on G' to find the backedge which is part of the cycle.
This runs in time $O(V+E) = O(V+V) = O(V)$.
Once we find the backedge, we can follow the parent references along the edges of the cycle, until we hit the other end of the backedge. These edges along with the back edge form a cycle. We can take the max. weight out of these edges. This represents the "bad" edge.



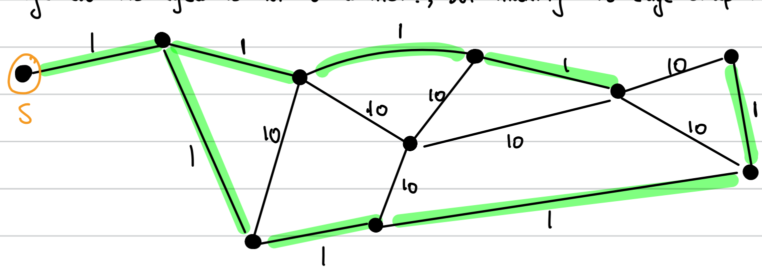
DFS:



⇒ Cycle. Biggest edge weight is 11, which is NOT part of MST.
∴ remove edge FE.

(b) If all edge weights are distinct, only 1 MST is possible.
Proof: Suppose (for contradiction) that there was more than one MST: T_1 and T_2, each of equal (minimum) weight. Then some edge e_1 in T_1 is NOT in T_2.
Similarly $\exists$ e_2 in T_2 that is not in T_1. Suppose $e_1 < e_2$. If we add edge e_1 to T_2 we create a cycle. The largest edge on this cycle is not in the MST., but making this edge swap means creating a smaller MST. Which is impossible!

(c)



Prims(S) = Dijkstras(S)

4a) Model as a graph with vertices as intersections and undirected edges as the bidirectional roads. The edge weights are the toll values, and u.tax is a vertex attribute.
Use a variation of Dijkstra's algorithm:

Initialise: Initialise $v.d = INF$ for all vertices.
Set $s.d = S.tax$
Insert all vertices into min Priority Queue.

Main Loop: While Q not empty:
$u = \text{ExtractMin}(Q)$
for all v in $Adj(u)$ where $v \in Q$
if $v.d > u.d + w(u,v) + v.tax$
DecreaseKey(Q, v, $u.d + w(u,v) + v.tax$)
$v.parent = u$

Return value: Print $t.d$
PrintBackwards(t)

Runtime: Dijkstra runs in $O(E\log V)$
$= O(m \log n)$

PrintBackwards(u)
If $u \neq NIL$
PrintBackwards(u.parent)
Print u.

4b) Let the 5 trail markers with toilets T1, T2, T3, T4, T5.

Step 1) Run Dijkstra's algorithm from S, storing distances in v.fromS
Any vertices for which v.fromS > 20, reset v.fromS = INF.

Step 2) Run Dijkstra's algorithm from T1, storing distance in v.fromT1
Any vertices for which v.fromT1 > 20, reset v.fromT1 = INF
Repeat for T2, T3, T4, T5.

Step 4) Create a new graph with vertex set S, F, T1, T2, T3, T4, T5.
The edges are directed and weighted. The weight of each edge corresponds to an attribute from the original graph:

$w(S, T1) = T1.fromS$
$w(S, T2) = T2.fromS$
$\vdots$
$w(T2, T1) = T1.fromT2$

Set all edge weights. Recall some edge weights will be ∞.

Step 5) Run Dijkstra's from S on the new graph, storing distances in v.d. The value in F.d represents the shortest distance from $S \to F$ while respecting the conditions.

Runtime: Each call to Dijkstra runs in $O(E\log V) = O(m\log n)$.
The last call to Dijkstra runs in constant time since the graph has a constant # of vertices.
$\therefore$ overall runtime is $O(m\log n)$